RX Family

Simple Flash API for RX

Introduction

RX Family MCUs are provided with flash memory for code storage (ROM) and flash memory for data storage (data flash). Both of these areas of flash memory can be programmed by user programs.

This application note presents an application programming interface (API) that can be utilized by developers to implement programming of flash memory for code storage (ROM) and flash memory for data storage (data flash) in user programs they create. This document describes how to use the API functions and how to import a project into a user application.

Target Devices

- RX610 Group
- RX621 Group, RX62N Group, RX62T Group, and RX62G Group
- RX630 Group, RX631 Group, RX63N Group, and RX63T Group
- RX210 Group, RX21A Group, and RX220 Group

Contents

1. Overview

In order to program the flash memory it is necessary to control the MCU's on-chip flash control unit (FCU) and execute a complex procedure. The API presented in this application note enables the user to program the flash memory without needing to worry about the complex procedure.

1.1 Supported Functionality

The API supports the following functionality.

- Programming the ROM and data flash
- API operating modes (blocking mode and non-blocking mode)
- Programming of ROM from ROM, and data flash from data flash
- Protection by means of lock bit

1.2 Programming Flash Memory

As shown in Figure 1.1 (left figure), the necessary code for programming the flash memory is located in the ROM. By running the code from the ROM, it is possible to program the target area in the flash memory (the data flash in this example), as shown in Figure 1.1 (right figure)

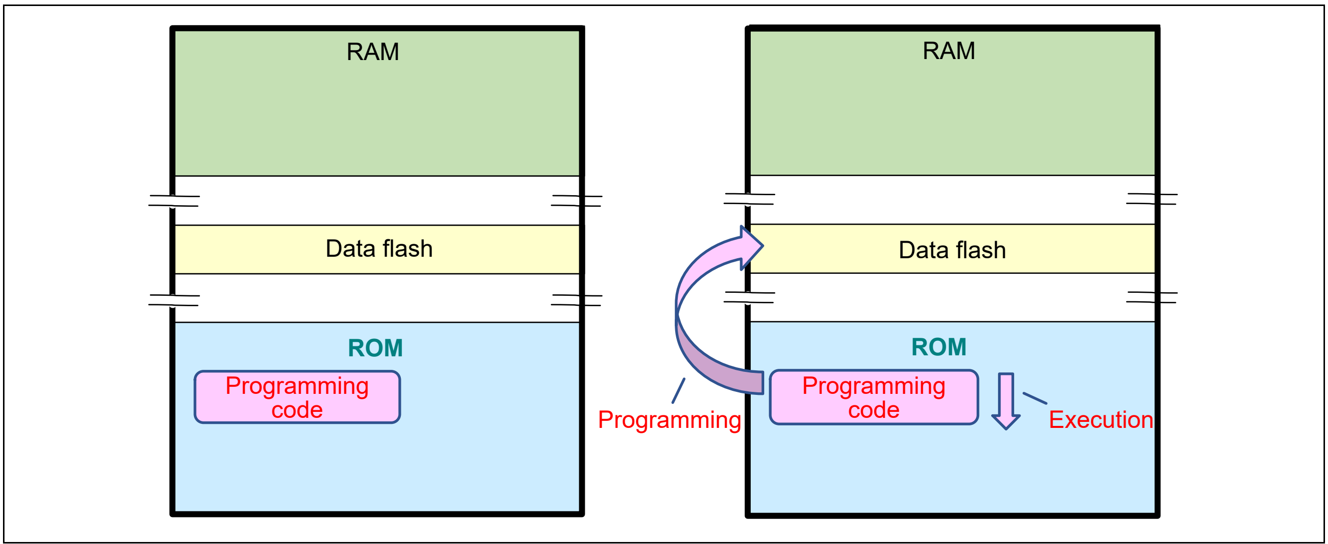


Figure 1.1 Location of Code for Programming Flash Memory and Programming Operation

1.3 API Operating Modes (Blocking Mode and Non-Blocking Mode)

The API has two operating modes: blocking mode and non-blocking mode. In blocking mode, after an API function is called, it does not return until processing of flash memory programming is complete. In non-blocking mode, after an API function is called, it returns before processing of flash memory programming finishes. For details of the API operating modes, refer to 2.6, Blocking Mode and Non-Blocking Mode.

1.4 Access Restrictions during API Execution

The FCU has a read mode for running programs and reading data, and a program/erase mode (P/E mode) for programming the flash memory. With the exception of a few API functions, the FCU transitions to P/E mode when an API function is run and programs the flash memory. While the FCU is in P/E mode, the types of read access to the flash memory listed in the table below are restricted. If the programming target area is read while in P/E mode, the read value is undefined.

Table 1.1 Read Access Restrictions

Programming Target Area (in P/E Mode)	Read Access Area			
	ROM	Data Flash	RAM	External Memory
ROM	×	○	○	○
Data flash	○	×	○	○

○: Access allowed.
×: Access prohibited.

For this reason, it is not possible to program the same area of the flash memory where the code necessary for programming is located, as shown in Figure 1.2.

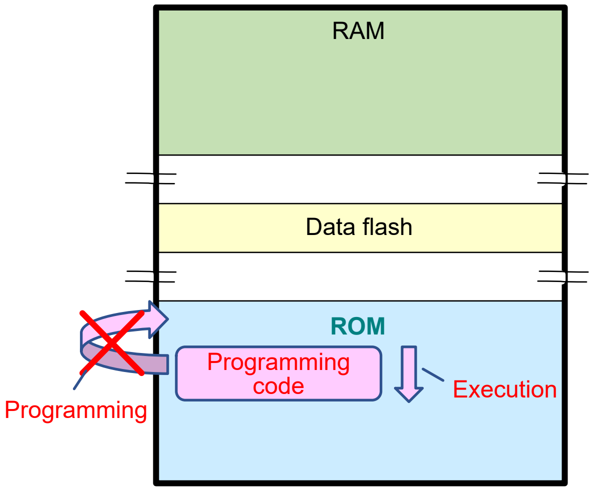


Figure 1.2 Programming Same Area of Flash Memory where Code Necessary for Programming Is Located

In order to program the ROM, it is necessary to run the program from an area other than the ROM, in other words from the RAM area (including external RAM). This also applies to the data that is programmed.

1.5 API Execution Area and Necessary Operations

In order to program the flash memory, it is necessary to run the API functions from an area other than the programming target area.

The default settings of the API assign the API functions to ROM areas (section P and section C), but it may be necessary to change the area from which the API functions run according to the API mode or programming target area. It may also be necessary to change the locations of associated links and interrupt vector tables, as well as preparing callback functions.

Table 1.2 Accommodating the API Mode

Programming Target Area	Mode	Possible Areas for API Execution and Interrupt Vector Tables
Data flash	Blocking mode	RAM (including external) ROM
	Non-blocking mode	
ROM	Blocking mode	RAM (including external)
	Non-blocking mode	

For information on running programs from the RAM, refer to 2.8, Running API Code from RAM. For information on moving interrupt vector tables, refer to 2.7, Interrupt Vector Tables and Interrupt Handlers.

In addition, it is necessary to prepare callback functions when using API functions in non-blocking mode. For information on preparing callback functions, refer to 2.6.2, Non-Blocking Mode.

2. API Information

2.1 Toolchain

The operation of the API has been confirmed with the following toolchain.

C/C++ Compiler Package for RX Family (CC-RX) V3.04.00

2.2 Header Files

To use the API with a user program, include the file r_flash_api_rx_if.h. In addition, in order to use the API you will need to make settings in r_flash_api_rx_config.h to match the user program.

2.3 Configuration

Settings for the API are made in r_flash_api_rx_config.h. Table 2.1 lists the configuration options in r_flash_api_rx_config.h.

Table 2.1 Configuration Items (r_flash_api_rx_config.h)

Configuration Options	
FLASH_MCU_xxxx	This option is defined to match the device used. Example: RX63N #define FLASH_MCU_RX63N
FLASH_API_RX_CFG_ICLK_HZ	Specifies the frequency of the system clock (ICLK) in Hz. Example: 100 MHz #define FLASH_API_RX_CFG_ICLK_HZ (100000000)
FLASH_API_RX_CFG_FCLK_HZ	RX610 Group or RX62x Group: Specifies the frequency of the peripheral module clock (PCLK) in Hz. RX63x Group or RX2x Group: Specifies the frequency of the FlashIF clock (FCLK) in Hz. Example: 50 MHz #define FLASH_API_RX_CFG_FCLK_HZ (50000000)
FLASH_API_RX_CFG_ROM_ SIZE_BYTES	Specifies the ROM size in bytes. A macro definition (SIZE_xB) can also be used to specify the size. (Defined in this header file.) Example: 2 MB #define FLASH_API_RX_CFG_ROM_SIZE_BYTES (SIZE_2MB)
FLASH_API_RX_CFG_DATA_ FLASH_SIZE_BYTES	Specifies the data flash size in bytes. A macro definition (SIZE_xB) can also be used to specify the size. (Defined in this header file.) Example: 32 KB #define FLASH_API_RX_CFG_ROM_SIZE_BYTES (SIZE_32KB)
FLASH_API_RX_CFG_ENABLE_ ROM_PROGRAMMING	ROM programming is enabled when this option is defined. To enable ROM programming, set this option as described in 2.8, Running API Code from RAM. Only data flash programming is enabled when this option is undefined.
FLASH_API_RX_CFG_FLASH_ TO_FLASH	ROM to ROM and data flash to data flash programming operations are enabled when this option is defined. If this option is defined, a RAM buffer for storing programming data is required. The size of the RAM buffer should be equivalent to the maximum amount of data that will be programmed to the ROM on the particular device.
FLASH_API_RX_CFG_DATA_ FLASH_BGO	Data flash programming in non-blocking mode is enabled when this option is defined. In this mode programming of the data flash takes place in the background, and API functions return before programming finishes. When programming finishes, a flash ready interrupt (FRDYI) is generated and a callback function is run. API functions do not return until programming is complete when this option is undefined.

Configuration Options	
FLASH_API_RX_CFG_ROM_BGO	ROM programming in non-blocking mode is enabled when this option is defined. In this mode programming of the ROM takes place in the background, and API functions return before programming finishes. When programming finishes, a flash ready interrupt (FRDYI) is generated and a callback function is run. API functions do not return until programming is complete when this option is undefined.
FLASH_API_RX_CFG_FLASH_ READY_IPL	Specifies the priority level of the flash ready interrupt. This setting takes effect when programming in non-blocking mode is enabled.
FLASH_API_RX_CFG_IGNORE_ LOCK_BITS	The lock bit protection functionality is disabled when this option is defined. When it is undefined, lock bit protection is enabled and attempts to program or erase blocks for which the lock bit is set fail.
FLASH_API_RX_CFG_COPY_ CODE_BY_API	The R_FlashCodeCopy function is enabled when this option is defined. When copying API functions located in the ROM section (PFRAM) to the RAM section (RPFRAM) by editing dbsct.c, without using the R_FlashCodeCopy function, leave this definition undefined (disabled).

2.4 Return Values

The return values of the API functions are listed below. These are defined in r_flash_api_rx_if.h. Some return values can have the same value, but no API function can have the same value defined for more than one return value at the same time.

```
/**** Function Return Values ****/
/* Operation was successful */
#define FLASH_SUCCESS           (0x00)
/* Flash area checked was blank, making this 0x00 as well to keep existing
   code checking compatibility */
#define FLASH_BLANK             (0x00)

/* The address that was supplied was not on aligned correctly for ROM or DF */
#define FLASH_ERROR_ALIGNED     (0x01)
/* Flash area checked was not blank, making this 0x01 as well to keep existing
   code checking compatibility */
#define FLASH_NOT_BLANK         (0x01)

/* The number of bytes supplied to write was incorrect */
#define FLASH_ERROR_BYTES       (0x02)
/* The address provided is not a valid ROM or DF address */
#define FLASH_ERROR_ADDRESS     (0x03)
/* Writes cannot cross the 1MB boundary on some parts */
#define FLASH_ERROR_BOUNDARY    (0x04)
/* Flash is busy with another operation */
#define FLASH_BUSY              (0x05)
/* Operation failed */
#define FLASH_FAILURE           (0x06)
/* Lock bit was set for the block in question */
#define FLASH_LOCK_BIT_SET      (0x07)
/* Lock bit was not set for the block in question */
#define FLASH_LOCK_BIT_NOT_SET  (0x08)
/* 'Address + number of bytes' for this operation went past the end of this
 * memory area. */
#define FLASH_ERROR_OVERFLOW (0x09)
```

2.5 Blocking Mode and Non-Blocking Mode

2.5.1 Blocking Mode

In blocking mode, after an API function is called, it does not return until processing of flash memory programming is complete. To run API functions in blocking mode, disable the following macro definitions in the configuration options.

- FLASH_API_RX_CFG_DATA_FLASH_BGO
- FLASH_API_RX_CFG_ROM_BGO

2.5.2 Non-Blocking Mode

In non-blocking mode, after an API function is called, it returns before processing of flash memory programming finishes. When flash memory programming finishes, a flash ready interrupt (FRDYI) is generated and a callback function is run from the interrupt handler routine.

To run API functions in non-blocking mode, enable the following macro definitions in the configuration options. Operation in non-blocking mode only occurs for processing with these macro definitions enabled.

- FLASH_API_RX_CFG_DATA_FLASH_BGO
- FLASH_API_RX_CFG_ROM_BGO

Only the following API functions operate in non-blocking mode. For all other API functions no flash ready interrupt (FRDYI) is generated and no callback function is run. (Operation is in blocking mode.)

- R_FlashErase function
- R_FlashEraseRange function
- R_FlashWrite function
- R_FlashDataAreaBlankCheck function

The following callback functions are run according to the API function. These callback functions must be prepared by the user, and they cannot be omitted.

- void FlashEraseDone(void)
  — This callback function is run when erasing of the ROM or data flash by the R_FlashErase function or R_FlashEraseRange function finishes.
- void FlashWriteDone(void)
  — This callback function is run when programming of the ROM or data flash by the R_FlashWrite function finishes.
- void FlashBlankCheckDone(uint8_t result)
  — This callback function is run when blank checking of the data flash by the R_FlashDataAreaBlankCheck function finishes. The result of the blank check is stored in the argument **result**. If the data flash is blank, the value stored is FLASH_BLANK, and if the data flash is not blank, the value stored is FLASH_NOT_BLANK.
- void FlashError(void)
  — This callback function is run when erasing, programming, or blank checking fails. Command lock or any applicable error status is canceled by the time the callback function runs, so the user does not need to run any processing targeting the FCU. You should implement whatever processing is appropriate for the user system.

2.6 Interrupt Vector Tables and Interrupt Handlers

Due to the restrictions on read access described in 1.4, Access Restrictions during API Execution, it is not possible to read correct data from the ROM after the FCU switches to P/E mode in order to program the flash memory. This restriction applies to everything stored in the ROM, including variable vector tables (interrupt vector tables), fixed vector tables, and interrupt handler code.

To enable interrupt handlers to run while in P/E mode, it is necessary to copy the interrupt vector tables and interrupt handler code to a RAM area (including external) in the same manner as the API functions and to make appropriate changes to the interrupt table register (INTB) values. Since it is not possible to change the location of fixed vector tables, care must be taken to ensure that no exception handling involving such tables occurs while in P/E mode.

The example below shows the method of copying an interrupt vector table to the RAM and changing the value of the interrupt table register (INTB). This is only an example, and you will need to study the appropriate method to use to match the user system.

```
/* RAM area where vector table is stored */
static uint32_t ram_vector_table[256];

/* Pointer for accessing interrupt vector table */
uint32_t *pvect_table;

/* Variable for loop */
uint16_t i;


/* == Relocation of interrupt vector table in RAM == */
pvect_table = (uint32_t *)__sectop("C$VECT");

for( i=0 ; i < 256 ; i++)
{
   ram_vect_table[i] = pvect_table[i]; /* Copy address of FRDYI interrupt
function */
}

set_intb((void *)ram_vect_table);
```

2.7 Running API Code from RAM

To run API code from the RAM, first prepare sections for storing the API code in both RAM (RPFRAM) and ROM (PFRAM), and then, after a reset, copy the code, including the API functions, from the ROM section (PFRAM) to the RAM section (RPFRAM).

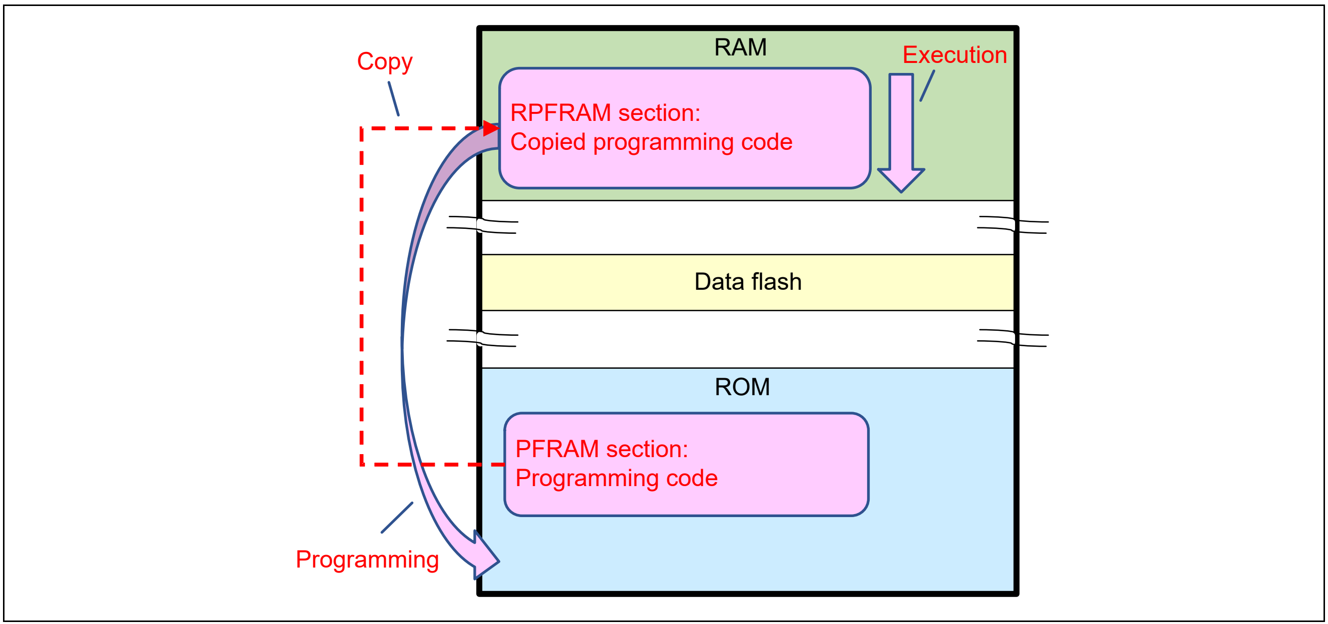


Figure 2.1 Running Code from RAM to Program ROM

Follow the steps below to make the necessary settings.

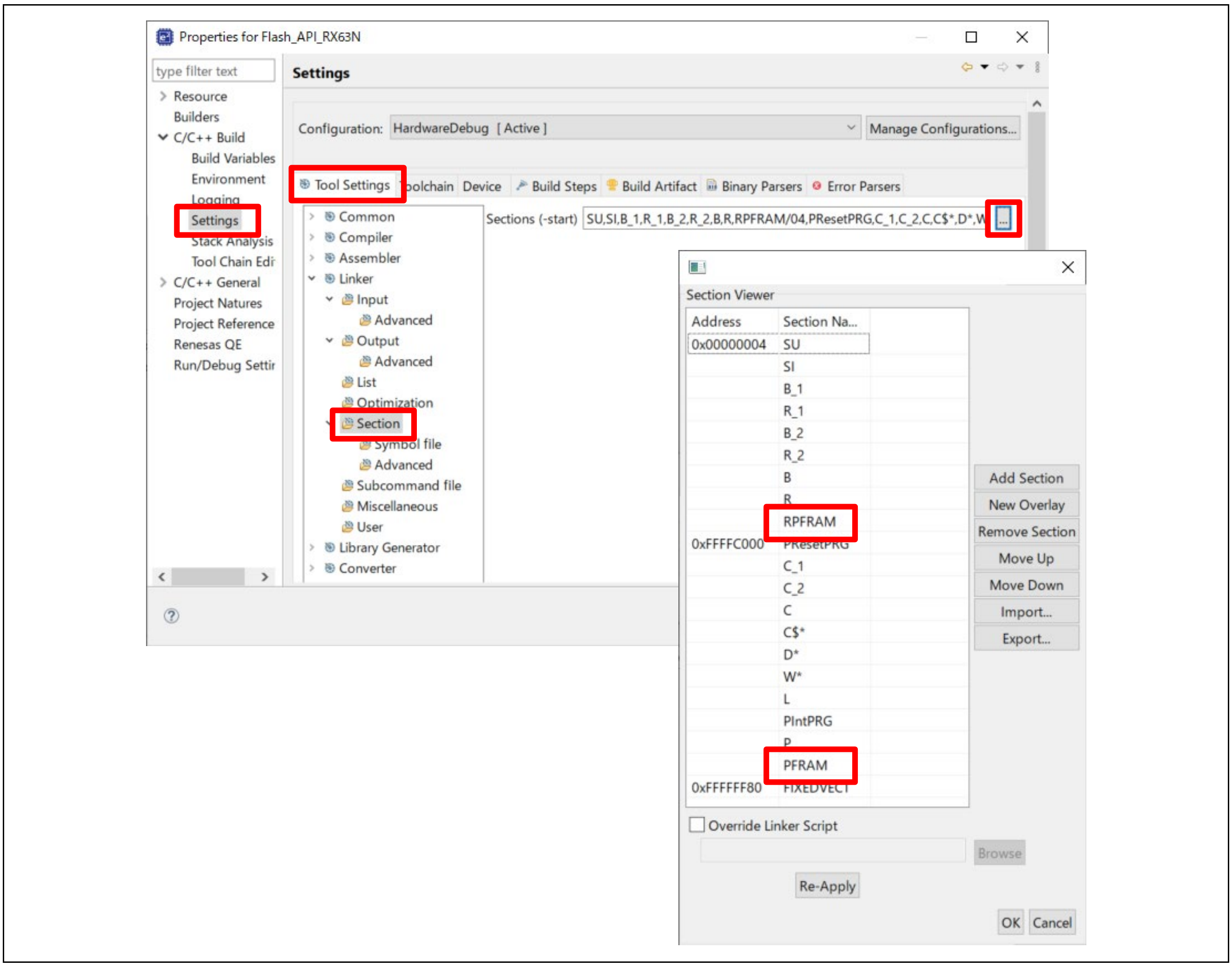
(1) Configuration option setting
Enable the following definition:
#define FLASH_API_RX_CFG_ENABLE_ROM_PROGRAMMING

(2) Adding sections

Add the **RPFRAM** section in the RAM area and the **PFRAM** section in the ROM area.

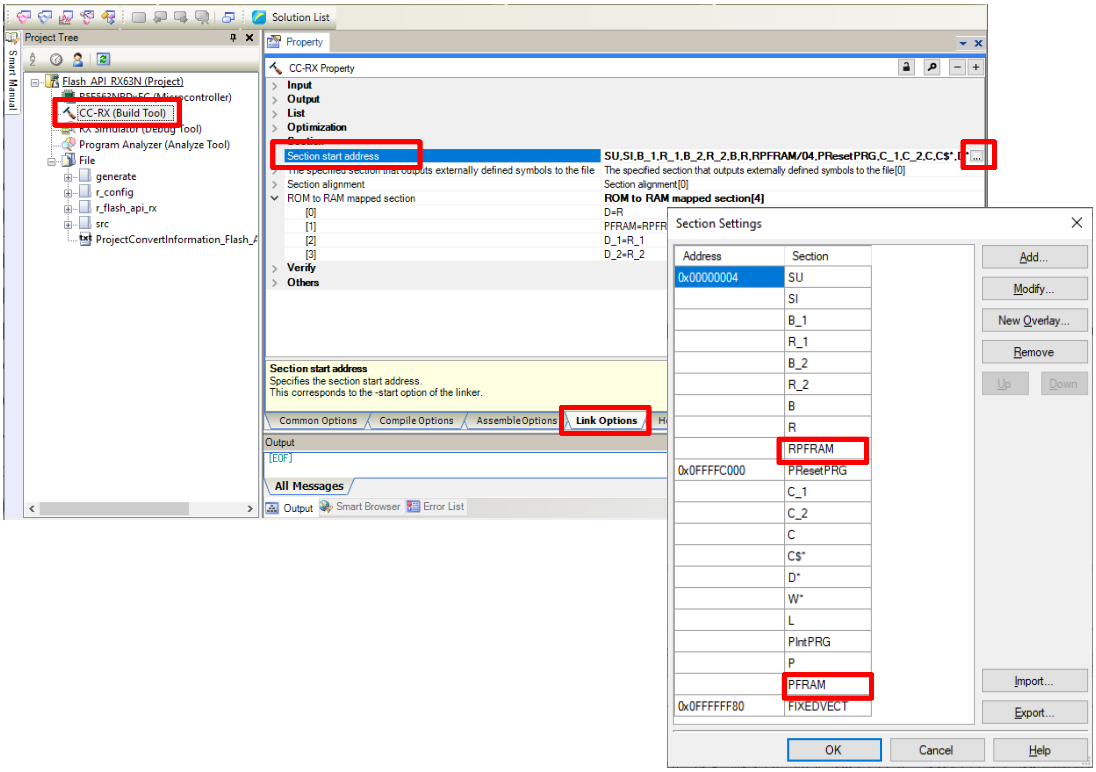
— e² studio

Select **Linker** → Section and click **...** to open the Section Viewer window. Add the PFRAM and RPFRAM sections as shown in the figure below.



— CS+

Select **CC-RX (Build Tool)** → Link Option → Section → **Section start address** and click ... to open the **Section Settings** window. Add the PFRAM and RPFRAM sections as shown in the figure below.

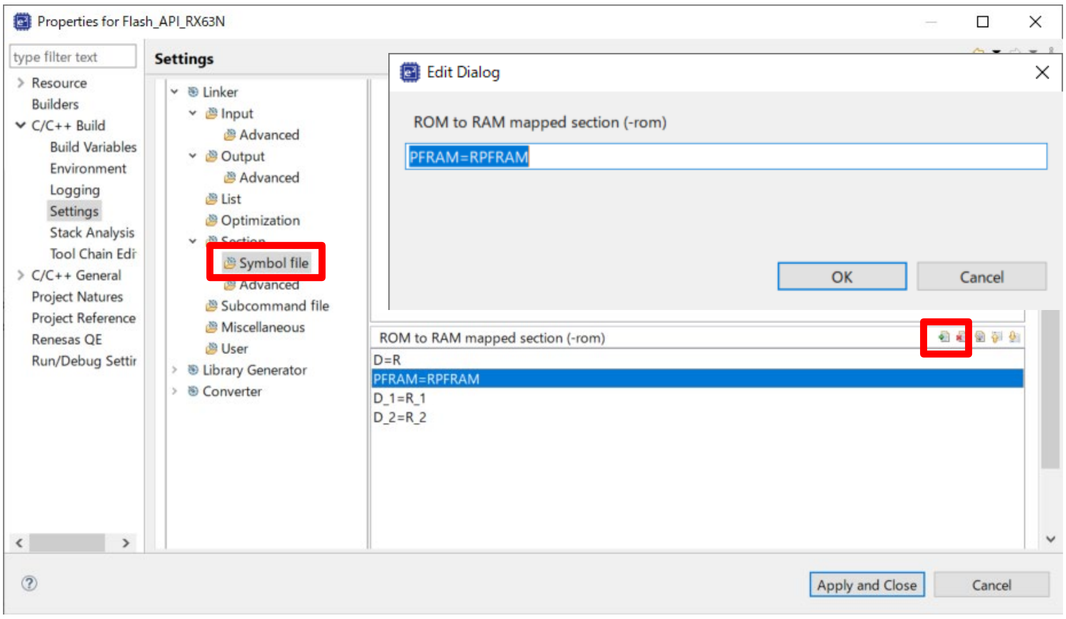


(3) Mapping from ROM to RAM

To create a linker map from the ROM section (PFRAM) to the RAM section (RPFRAM), add the following item under **`ROM to RAM mapped section`**.

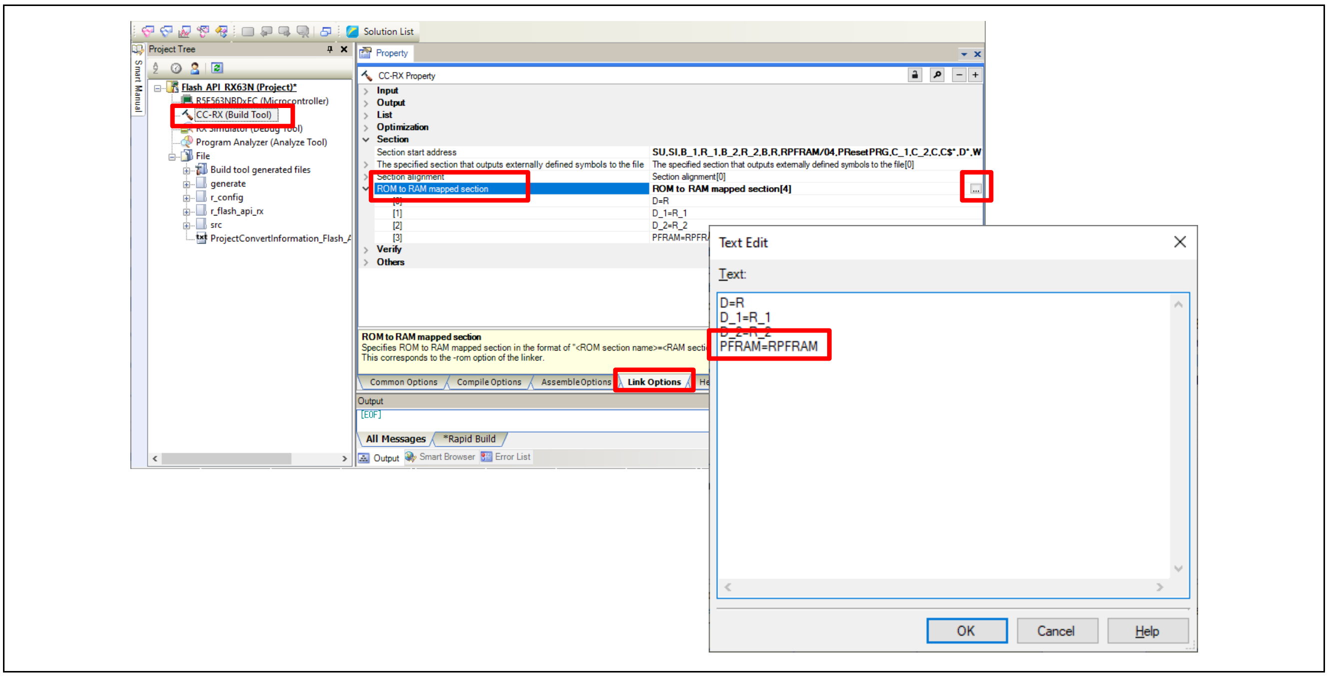
— e² studio

Select **Linker** → Section → Symbol file and click the Add button under **`ROM to RAM mapped section`** to open the window shown below. Add **PFRAM=RPFRAM** as shown in the figure below.



— CS+

Select **CC-RX(Build Tool)** → Link Option → **Section** → **ROM to RAM mapped section** and click ... to open the window shown below. Add **PFRAM=RPFRAM** as shown in the figure below.



(4) Transferring program code from ROM to RAM after reset

Transfer the program code from the ROM section (PFRAM) to the RAM section (RPFRAM). Either of the following two methods can be used to copy the program code to the RAM section (RPFRAM).

1. Editing the dbsct.c file

   The dbsct.c file specifies the areas to be initialized after a reset. Add code to transfer the program code from the PFRAM section to the RPFRAM section as shown in red text below.

```
-- FILE [dbsct.c] --
#pragma section $DSEC
static const struct {
    _UBYTE *rom_s; /* Initial address on ROM of initialization data section */
    _UBYTE *rom_e; /* Final address on ROM of initialization data section */
    _UBYTE *ram_s; /* Initial address on RAM of initialization data section */
}   DTBL[] = {
    { __sectop("D"), __secend("D"), __sectop("R") } ,
    { __sectop("PFRAM"), __secend("PFRAM"), __sectop("RPFRAM") }
};
```

2. Running the R_FlashCodeCopy function

   The purpose of the R_FlashCodeCopy function is to transfer the program code to the RAM section (RPFRAM). Call this function from the user program before calling any other API functions. It is necessary to define FLASH_API_RX_CFG_COPY_CODE_BY_API in r_flash_api_rx_config.h in order to use the R_FlashCodeCopy function.

2.8 Programing from ROM to ROM or from Data Flash to Data Flash

This functionality is implemented in software. It is not built into the FCU.

In the default configuration option settings FLASH_API_RX_CFG_FLASH_TO_FLASH is left undefined, so it is not possible to specify an address in the write destination area as the write source address (buffer_addr) specified by the second argument of the R_FlashWrite function.

However, enabling the FLASH_API_RX_CFG_FLASH_TO_FLASH definition in the configuration options makes it possible to specify an address in the write destination area as the write source address (buffer_addr). For example, you can specify an address in the ROM when the write destination area is the ROM, and you can specify an address in the data flash when the write destination area is the data flash. For details of the R_FlashWrite function, refer to 3.4, R_FlashWrite.

When the R_FlashWrite function is run, this functionality stores in the RAM the data to be written before the transition to P/E mode, and uses the data stored in the RAM to program the flash memory after the transition to P/E mode. Therefore, enabling the FLASH_API_RX_CFG_FLASH_TO_FLASH definition causes the data to be written to be maintained in the RAM buffer set aside for storing programming data. The size of the RAM buffer should be equivalent to the maximum amount of data that will be programmed to the ROM on the particular device.

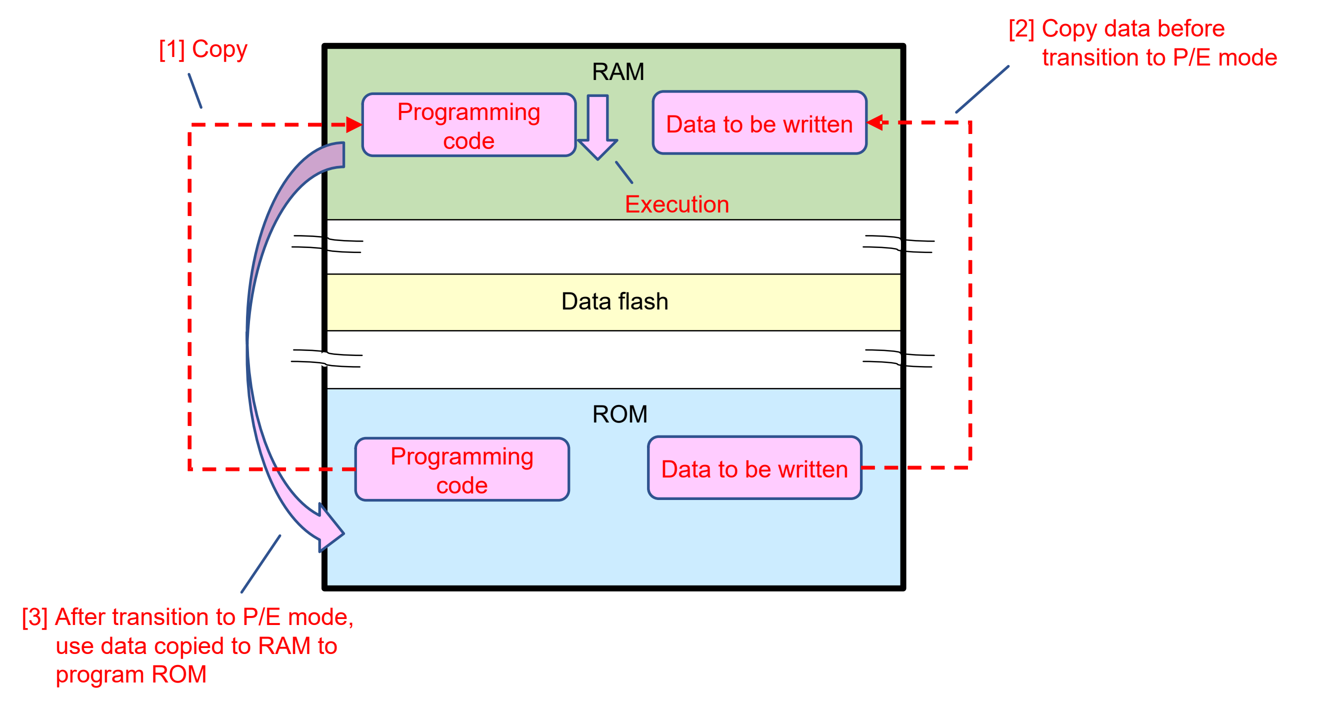


Figure 2.2 Example of Programming from ROM to ROM

2.9 Usage Precautions

2.9.1 Operating Frequency when Running API Functions

When running API functions, set the operating frequency within the range shown in Table 2.2. Set the configuration option FLASH_API_RX_CFG_FCLK_HZ to the same value as the operating frequency.

Table 2.2 Operating Frequency

Device	FCU Clock Source	Frequency Range
RX610 Group RX62x Group	Peripheral module clock (PCLK)	8 MHz to PCLK max. frequency
RX63x Group RX2x Group	FlashIF clock (FCLK)	4 MHz to FCLK max. frequency

2.9.2 Accessing Data Flash after Reset

Accessing the data flash (reading, programming, or erasing) is prohibited in the initial state after a reset. To access the data flash, it is first necessary to enable access by running the R_FlashDataAreaAccess function. For details, refer to 3.5, R_FlashDataAreaAccess.

2.9.3 Flash Memory Value after Erase

After erasure, the values of the ROM and data flash differ. The read value of the ROM after erasure is FFh, but that of the data flash is undefined. To determine whether or not the data flash is blank, run the R_FlashDataAreaBlankCheck function.

2.9.4 Block Address Constants

The API makes use of constant array g_flash_BlockAddresses[] containing const type data. This array defines the start addresses of the blocks in the flash memory. Note that addresses in the ROM are defined as P/E mode addresses and cannot be used for read accesses. Also, the array is stored in a ROM area (section C) by default. Note that the array will be deleted if the ROM is erased. The array is defined in the header file for the specific device (r_flash_api_rxXXX.h), stored in r_flash_api_rx\src\targets.

```
/* Data Structure */
const uint32_t g_flash_BlockAddresses[86] = {
   0x00FFF000,  /* EB00 */
   0x00FFE000,  /* EB01 */
   0x00FFD000,  /* EB02 */
   0x00FFC000,  /* EB03 */
   ...
};
```

2.9.5 Programming across Boundaries in ROM Area

Some RX Family products have multiple ROM areas. For example, RX63N Group products with 2 MB of ROM have four ROM areas (areas 0, 1, 2, and 3). The programming that can be performed by an API function each time it is run is limited to a single area and cannot cross a boundary between ROM areas. To program data across ROM area boundaries, divide the data to be written into segments and program each one separately.

Refer to Table 2.3 for the ROM areas of each product.

Table 2.3 ROM Area Boundaries

ROM Capacity	Address	RX610	RX62N RX621	RX62T RX62G	RX630 RX63N RX631	RX63T	RX210	RX220	RX21A
2 MB	FFE0 0000h	Area 1			Area 3				
1.5 MB	FFE8 0000h				Area 2				
1 MB	FFF0 0000h	Area 0			Area 1		Area 1		
512 KB	FFF8 0000h		Area 0		Area 0	Area 0	Area 0		Area 0
256 KB	FFFC 0000h			Area 0				Area 0	

2.10 Memory Usage

The amount of ROM and RAM used differs according to the which configuration options are enabled. For example, when ROM programming (FLASH_API_RX_CFG_ENABLE_ROM_PROGRAMMING) is enabled, areas for storing program code are necessary in both the RAM and ROM since API functions for programming the ROM are run from the RAM.

Table 2.4 lists several configuration patterns, and Table 2.5 lists the memory usage for each configuration pattern.

Table 2.4 Configuration Patterns

Configuration Options		Option Enabled/Disabled
1	Programming data flash only	
	FLASH_API_RX_CFG_ENABLE_ROM_PROGRAMMING	Disabled
	FLASH_API_RX_CFG_FLASH_TO_FLASH	Disabled
	FLASH_API_RX_CFG_DATA_FLASH_BGO	Enabled
	FLASH_API_RX_CFG_ROM_BGO	Disabled
	FLASH_API_RX_CFG_IGNORE_LOCK_BITS	Enabled
	FLASH_API_RX_CFG_COPY_CODE_BY_API	Disabled
2	Programming ROM	
	FLASH_API_RX_CFG_ENABLE_ROM_PROGRAMMING	Enabled
	FLASH_API_RX_CFG_FLASH_TO_FLASH	Disabled
	FLASH_API_RX_CFG_DATA_FLASH_BGO	Disabled
	FLASH_API_RX_CFG_ROM_BGO	Disabled
	FLASH_API_RX_CFG_IGNORE_LOCK_BITS	Disabled
	FLASH_API_RX_CFG_COPY_CODE_BY_API	Enabled
3	Maximum ROM/RAM usage	
	FLASH_API_RX_CFG_ENABLE_ROM_PROGRAMMING	Enabled
	FLASH_API_RX_CFG_FLASH_TO_FLASH	Enabled
	FLASH_API_RX_CFG_DATA_FLASH_BGO	Enabled
	FLASH_API_RX_CFG_ROM_BGO	Enabled
	FLASH_API_RX_CFG_IGNORE_LOCK_BITS	Disabled
	FLASH_API_RX_CFG_COPY_CODE_BY_API	Enabled

Compiler option settings (common)

- Language specification: C89
- Optimization level: Level 2

Table 2.5 Memory Usage

Device	Configuration Pattern	ROM (Bytes)	RAM (Bytes)
RX610	1	1850	19
	2	2039	1300
	3	2554	2409
RX62N	1	1926	19
	2	2094	1247
	3	2609	2376
RX62G, RX62T	1	1862	19
	2	2030	1247
	3	2545	2376
RX630, RX63N	1	2302	19
	2	2496	1526
	3	3051	2566
RX63T	1	2054	19
	2	2189	1387
	3	2744	2427
RX210	1	3993	19
	2	4146	1469
	3	4702	2494
RX220	1	2378	19
	2	2508	1411
	3	3064	2432
RX21A	1	2896	19
	2	3026	1413
	3	3582	2438

2.11 Importing API Functions into a User Project

Follow the steps below to import the API functions into your user project.

1. Copy the entire contents of the **r_flash_api_rx** directory in the source directory to the user project.
2. Add the file **r_flash_api_rx\src\r_flash_api_rx.c** as a build target.
3. Add the **r_flash_api_rx** directory to the include path.
4. Add the **r_flash_api_rx\src** directory to the include path.
5. In the **r_flash_api_rx\src\targets** directory, delete the directories with names matching products that will not be used on the user system.
6. Copy **r_flash_api_rx_config_reference.h** from the **ref** directory to the user project directory, and rename the file as **r_flash_api_rx_config.h**.
7. Add the directory containing the file **r_flash_api_rx_config.h** to the include path.
8. Configure **r_flash_api_rx_config.h** to match the user system.
9. Add lines to the file **r_flash_api_rx_if.h** to include the source files used by the API functions.

Note: Deleting the **r_flash_api_rx\ref** directory from the user project will have no effect on the build process.

To program the ROM, proceed with the steps listed in 2.8, Running API Code from RAM.

3. API Functions

3.1 Overview

Table 3.1 is an overview of the API functions.

Table 3.1 List of API Functions

Function	Description
R_FlashErase	Erases the target block in the flash memory.
R_FlashEraseRange	Erases the blocks within the target range of the data flash. (Not supported on RX610 Group and RX62x Group.)
R_FlashWrite	Programs data to the flash memory.
R_FlashDataAreaAccess	Enables or disables access to the data flash (reading, programming, and erasing).
R_FlashDataAreaBlankCheck	Performs a blank check on the data flash.
R_FlashProgramLockBit	Sets the lock bit for the target block in the ROM, prohibiting erasing and programming.
R_FlashReadLockBit	Reads the status of the lock bit for the target block in the ROM.
R_FLashSetLockBitProtection	Enable or disable the lock bit protection functionality.
R_FlashGetStatus	Returns the API processing status.
R_FlashCodeCopy	Copies the API code from the ROM section (PFRAM) to the RAM section (RPFRAM).

3.2 R_FlashErase

Erases the target block in the flash memory.

Format

```
uint8_t R_FlashErase(uint32_t block);
```

Parameters

block Specifies the block number to be erased. Block numbers are defined in the r_flash_api_rxXXX.h file corresponding to the device, located in r_flash_api_rx/src/targets. For example, on the RX610 the block at address 0xFFFFE000 is referenced as block 0 in the user's manual and would be specified by this parameter as BLOCK_0.

Return Values

FLASH_SUCCESS: Normal end (In non-blocking mode, this indicates that processing of the flash memory started normally.)

FLASH_FAILURE: Abnormal end (Processing attempted on a ROM area for which the lock bit is set or a data flash area for which the access is prohibited; or processing timeout.)

FLASH_BUSY: Other processing of the flash memory is currently in progress.

FLASH_ERROR_ADDRESS: Invalid block number

Description

Erases the block specified by the argument. Block sizes differ according to the device group as well as the flash memory area on the device. In addition, the data flash on some devices has very small block sizes defined, so the function erases multiple blocks at once. For details of the block structure, refer to the user's manual of the specific device.

Table 3.2 lists the erase size for each device.

Table 3.2 Erase Size

Device	ROM	Data Flash
RX610 Group	128 KB, 64 KB, 8 KB	8 KB
RX62x Group	16 KB, 4 KB	2 KB
RX630, RX63N, RX631 Group	64 KB, 32 KB, 16 KB, 4 KB	2 KB (32 bytes × 64 blocks)
RX63T Group	16 KB, 4 KB	2 KB (32 bytes × 64 blocks)
RX2x Group	2 KB	2 KB (128 bytes × 16 blocks)

Reentrant

No.

Example

```
uint32_t loop;
uint8_t ret;

/* Specify the erase block */
ret = R_FlashErase(BLOCK_0);

/* Check for errors. */
if (FLASH_SUCCESS != ret)
{
    . . .
}

/* Erase multiple blocks (erase block 0 to block "NUM_BLOCKS_TO_ERASE")*/
for (loop = 0; loop < NUM_BLOCKS_TO_ERASE; loop++)
{
    /* Erase block */
    ret = R_FlashErase(loop);

    /* Check for errors. */
    if (FLASH_SUCCESS != ret)
    {
        . . .
    }
}
```

Special Notes

To erase a block in the data flash, first run the R_FlashDataAreaAccess function to enable access to the data flash.

3.3 R_FlashEraseRange (Not supported on RX610 Group and RX62x Group.)

Erases the blocks within the target range of the data flash.

Format

```
uint8_t R_FlashEraseRange(uint32_t start_addr, uint32_t bytes);
```

Parameters

start_addr Specifies the block start address of the target range to be erased. The address must be in alignment with the block size. Refer to Table 3.3 for the method of calculating block sizes and addresses on each device.

bytes Specifies the number of bytes to be erased. This value must be a multiple of the data flash block size. For example, on the RX630, valid setting values are 32, 64, 96, and so on because the data flash block size is 32 bytes.

Return Values

FLASH_SUCCESS: Normal end (In non-blocking mode, this indicates that processing of the flash memory started normally.)

FLASH_FAILURE: Abnormal end (Processing attempted on a data flash area for which the access is prohibited; or processing timeout.)

FLASH_BUSY: Other processing of the flash memory is currently in progress.

FLASH_ERROR_BYTES: Byte count is not a multiple of the block size.

FLASH_ERROR_ADDRESS: Invalid address

FLASH_ERROR_ALIGNED: Block start address not specified.

FLASH_ERROR_OVERFLOW: Erase range exceeds data flash area.

Description

Erases blocks within the specified range (start_addr to (start_addr + bytes)).

Table 3.3 Data Flash Block Sizes

Device	Block Size	Method of Calculating Block N Address
RX63x Group	32 bytes (32 bytes × 1,024 blocks = 32 KB)	N × 32 + start address of data flash area (0010 0000h)
RX2x Group	128 bytes 128 bytes × 64 blocks = 8 KB)	N × 128 + start address of data flash area (0010 0000h)

Reentrant

No.

Example

```
uint8_t ret;

/* Erase 64 bytes. */
ret = R_FlashEraseRange(address, 64);

/* Check for errors. */
if (FLASH_SUCCESS != ret)
{
    . . .
}
```

Special Notes

- This function is not supported on the RX610 Group and RX62x Group. Use the R_FlashErase function to perform erasures instead.
- This function can only be used to erase the data flash.
- To use this function, first run the R_FlashDataAreaAccess function to enable access to the data flash.

3.4 R_FlashWrite

Programs data to the flash memory.

Format

```
uint8_t R_FlashWrite(uint32_t   flash_addr,
                     uint32_t   buffer_addr,
                     uint16_t   bytes);
```

Parameters

flash_addr   Specifies the write destination address. The address must be aligned with the minimum write size.

buffer_addr   Specifies the write source address.

bytes   Specifies the byte count of the data to be written. This value must be a multiple of the minimum write size. Refer to Table 3.4 for the minimum write size on each device.

Return Values

FLASH_SUCCESS: Normal end (In non-blocking mode, this indicates that processing of the flash memory started normally.)

FLASH_FAILURE: Abnormal end (Processing attempted on an area that was not blank or a ROM area for which the lock bit is set or data flash area for which the access is prohibited; or processing timeout..)

FLASH_BUSY: Other processing of the flash memory is currently in progress.

FLASH_ERROR_ALIGNED: Address is not aligned with minimum write size.

FLASH_ERROR_BYTES: Byte count is not a multiple of the minimum write size.

FLASH_ERROR_ADDRESS: Invalid address

FLASH_ERROR_BOUNDARY: Writes are not allowed to cross boundaries between ROM areas.

FLASH_ERROR_OVERFLOW: Write range exceeds ROM or data flash area.

Description

Programs data to the flash memory. The write address must be aligned with the minimum write size. Also, the byte count must be a multiple of the minimum write size. The minimum write size differs according to the device, as shown in the table below.

Table 3.4 Minimum Write Sizes

Device	ROM	Data Flash
RX610 Group and RX62x Group	256 bytes	8 bytes, 128 bytes
RX63x Group	128 bytes	2 bytes
RX210 Group	2 bytes, 8 bytes, 128 bytes	2 bytes, 8 bytes

Some RX Family products have multiple ROM areas. The programming that can be performed by the R_FlashWrite function each time it is run cannot cross a boundary between ROM areas. To program data across ROM area boundaries, divide the data to be written into segments and run the R_FlashWrite function for each one separately. For details on ROM areas, refer to2.10.5,Programming across Boundaries in ROM Area.

In addition, enabling the FLASH_API_RX_CFG_FLASH_TO_FLASH definition in r_flash_api_rx_config.h makes it possible to specify a write source address in the same area as the write destination address. For example, you can specify an address in the ROM when the write destination area is the ROM, and you can specify an address in the data flash when the write destination area is the data flash. For details, refer to 2.9, Programing from ROM to ROM or from Data Flash to Data Flash.

Reentrant

No.

Example

```
uint8_t ret;
uint8_t write_buffer[PROGRAM_SIZE] = “Hello World...”;

/* Write data to internal memory. */
ret = R_FlashWrite(address, (uint32_t)write_buffer, PROGRAM_SIZE);

/* Check for errors. */
if (FLASH_SUCCESS != ret)
{
    . . .
}
```

Special Notes

To program the data flash, first run the R_FlashDataAreaAccess function to enable access to the data flash.

3.5 R_FlashDataAreaAccess

Enables or disables access to the data flash (reading, programming, and erasing). After a reset, run the R_FlashDataAreaAccess function to enable access to the data flash (reading, programming, and erasing).

Format

```
void R_FlashDataAreaAccess(uint16_t read_en_mask,
                           uint16_t write_en_mask);
```

Parameters

read_en_mask Enables or disables read access for the blocks corresponding to the various bits. Setting a bit to 1 enables read access to the corresponding block, and clearing a bit to 0 disables read access to the corresponding block. Table 3.5 lists the blocks corresponding to the various bits.

write_en_mask Enables or disables write or erase access for the blocks corresponding to the various bits. Setting a bit to 1 enables write or erase access to the corresponding block, and clearing a bit to 0 disables write or erase access to the corresponding block. Table 3.5 lists the blocks corresponding to the various bits.

Return Values

None.

Description

Enables or disables access to the data flash. Before accessing the data flash, run the R_FlashDataAreaAccess function to enable access to the data flash.

Table 3.5 lists the blocks corresponding to the various bits. A dash (—) indicates that the bit setting has no effect.

Table 3.5 Correspondence of Bits to Blocks

Device	Size	Block Structure	read_en_mask, write_en_mask							
			b15	**b14**	**b13**	**b12**	**b11**	**b10**	**b9**	**b8**
			b7	**b6**	**b5**	**b4**	**b3**	**b2**	**b1**	**b0**
RX610	32 KB	8 KB × 4 blocks	—	—	—	—	—	—	—	—
			—	—	—	—	DB3	DB2	DB1	DB0
RX621 RX62N	32 KB	2 KB × 16 blocks	DB15	DB14	DB13	DB12	DB11	DB10	DB9	DB8
			DB7	DB6	DB5	DB4	DB3	DB2	DB1	DB0
RX62T RX62G	8 KB	2 KB × 4 blocks	—	—	—	—	—	—	—	—
			—	—	—	—	DB3	DB2	DB1	DB0
	32 KB	2 KB × 16 blocks	DB15	DB14	DB13	DB12	DB11	DB10	DB9	DB8
			DB7	DB6	DB5	DB4	DB3	DB2	DB1	DB0
RX630 RX631 RX63N	32 KB	32 bytes × 1,024 blocks	960 to 1,023	896 to 959	832 to 895	768 to 831	704 to 767	640 to 703	576 to 639	512 to 575
			448 to 511	384 to 447	320 to 383	256 to 319	192 to 255	128 to 191	64 to 127	0 to 63
RX63T	8 KB	32 bytes × 256 blocks	—	—	—	—	—	—	—	—
			—	—	—	—	192 to 255	128 to 191	64 to 127	0 to 63
	32 KB	32 bytes × 1,024 blocks	960 to 1,023	896 to 959	832 to 895	768 to 831	704 to 767	640 to 703	576 to 639	512 to 575
			448 to 511	384 to 447	320 to 383	256 to 319	192 to 255	128 to 191	64 to 127	0 to 63
RX210 RX220 RX21A	8 KB	128 bytes × 64 blocks	—	—	—	—	—	—	—	—
			—	—	—	—	DB48 to DB63	DB32 to DB47	DB16 to DB31	DB00 to DB15

Reentrant

No.

Example

```
/* Enable reading, writing, and erasing of all data flash blocks. */
R_FlashDataAreaAccess(0xFFFF, 0xFFFF);
```

Special Notes

None.

3.6 R_FlashDataAreaBlankCheck

Performs a blank check on the data flash.

Format

```
uint8_t R_FlashDataAreaBlankCheck(uint32_t address,
                                  uint8_t size);
```

Parameters

address Specifies the address or block number to undergo a blank check. When specifying an address, the value should align with the blank check size specified by size. When specifying a block number, use the value defined in the r_flash_api_rxXXX.h file corresponding to the device, located in r_flash_api_rx/src/targets.

size Specifies the size of the blank check. The following two values may be specified.

BLANK_CHECK_SMALLEST: Minimum blank check size

BLANK_CHECK_ENTIRE_BLOCK: Multiple-block size

Return Values

FLASH_BLANK: (In blocking mode)
The data flash area is blank.

(In non-blocking mode with size set to BLANK_CHECK_SMALLEST)
The data flash area is blank.

(In non-blocking mode with size set to BLANK_CHECK_ENTIRE_BLOCK)
Blank check processing started normally.

FLASH_NOT_BLANK: The data flash area is not blank.

FLASH_FAILURE: Abnormal end (Normally not returned.)

FLASH_BUSY: Other processing of the flash memory is currently in progress.

FLASH_ERROR_ADDRESS: Invalid address

FLASH_ERROR_BYTES: Invalid size

FLASH_ERROR_ALIGNED: Address is not aligned with blank check size.
(Applies only when size = BLANK_CHECK_SMALLEST.)

Description

Performs a blank check on the data flash. The R_FlashDataAreaBlankCheck function can be used on the data flash only. The blank check size can be specified as BLANK_CHECK_SMALLEST (minimum blank check size) or BLANK_CHECK_ENTIRE_BLOCK (multiple-block size). As shown in Table 3.6, the blank check sizes differ by device.

Table 3.6 Blank Check Sizes

Device	Blank Check Size
RX610	BLANK_CHECK_SMALLEST: 8 bytes
	BLANK_CHECK_ENTIRE_BLOCK: 1 block (8 KB)
RX62x	BLANK_CHECK_SMALLEST: 8 bytes
	BLANK_CHECK_ENTIRE_BLOCK: 1 block (2 KB)
RX63x	BLANK_CHECK_SMALLEST: 2 bytes
	BLANK_CHECK_ENTIRE_BLOCK: 64 blocks (2 KB)
RX210	BLANK_CHECK_SMALLEST: 2 bytes
	BLANK_CHECK_ENTIRE_BLOCK: 16 blocks (2 KB)

Reentrant

No.

Example

```
uint8_t ret;

/* Blank check a small data flash address. Blocking mode operation Only */
ret = R_FlashDataAreaBlankCheck(address, BLANK_CHECK_SMALLEST);

/* Check result. */
if (FLASH_NOT_BLANK == ret)
{
    /* Block is not blank. */
    . . .
}
else if (FLASH_BLANK == ret)
{
    /* Block is blank. */
    . . .
}

/* Blank check an entire data flash block. */
ret = R_FlashDataAreaBlankCheck(BLOCK_DB0, BLANK_CHECK_ENTIRE_BLOCK);

/* Check result. */
if (FLASH_NOT_BLANK == ret)
{
    /* Block is not blank. */
    . . .
}
else if (FLASH_BLANK == ret)
{
    /* Block is blank. */
    . . .
}
```

Special Notes

When the second argument (size) is set to BLANK_CHECK_SMALLEST (minimum blank check size), the blank check is performed in blocking mode even if non-blocking mode is enabled because the processing completes more quickly than writing or erasure. When it is set to BLANK_CHECK_ENTIRE_BLOCK (multiple block size), the enabled API operating mode is used.

3.7 R_FlashProgramLockBit

Sets the lock bit for the target block in the ROM, prohibiting erasing and programming.

Format

```
uint8_t R_FlashProgramLockBit(uint32_t block);
```

Parameters

block Specifies the block number for which the lock bit is set. Block numbers are defined in the r_flash_api_rxXXX.h file corresponding to the device, located in r_flash_api_rx/src/targets. For example, on the RX610 the block at address 0xFFFFE000 is referenced as block 0 in the user's manual and would be specified by this parameter as BLOCK_0.

Return Values

FLASH_SUCCESS:	Normal end
FLASH_FAILURE:	Abnormal end (Processing attempted on area for which lock bit is already set.)
FLASH_BUSY:	Other processing of the flash memory is currently in progress.
FLASH_ERROR_ADDRESS:	Invalid block number

Description

Sets the lock bit for the specified block.

Reentrant

No.

Example

```
uint8_t ret;

/* Enable lock bit protection (this is default out of reset) */
ret = R_FlashSetLockBitProtection(true);

/* Check for errors. */
if (FLASH_SUCCESS != ret)
{
    . . .
}

/* Program lock bits */
ret = R_FlashProgramLockBit(flash_block);

/* Check for errors. */
if (FLASH_SUCCESS != ret)
{
    . . .
}
```

Special Notes

- If a block is erased while the lock bit protection functionality is disabled, the lock bit corresponding to that block is cleared.
- To use the R_FlashProgramLockBit function, disable the FLASH_API_RX_CFG_IGNORE_LOCK_BITS definition in r_flash_api_rx_config.h.

3.8 R_FlashReadLockBit

Reads the status of the lock bit for the target block in the ROM.

Format

```
    uint8_t R_FlashReadLockBit(uint32_t block);
```

Parameters

block Specifies the block number for which the lock bit status is read.

Return Values

FLASH_LOCK_BIT_SET:	The lock bit has been set.
FLASH_LOCK_BIT_NOT_SET:	The lock bit has been cleared.
FLASH_FAILURE:	Abnormal end (Normally not returned.)
FLASH_BUSY:	Other processing of the flash memory is currently in progress.
FLASH_ERROR_ADDRESS:	Invalid block number

Description

Reads the status of the lock bit for the specified block. The return value is determined by the read result. A return value of FLASH_LOCK_BIT_SET (0x07) indicates that the lock bit of the target block has been set. A return value of FLASH_LOCK_BIT_NOT_SET (0x08) indicates that the lock bit of the target block has been cleared.

Reentrant

No.

Example

```
uint8_t ret;

/* Program lock bits */
ret = R_FlashReadLockBit(flash_block);

/* Check result. */
if (FLASH_LOCK_BIT_SET == ret)
{
    /* Lock bit is set for this block. */
    . . .
}
else if (FLASH_LOCK_BIT_NOT_SET == ret)
{
    /* Lock bit was not set for this block. */
    . . .
}
```

Special Notes

- If a block is erased while the lock bit protection functionality is disabled, the lock bit corresponding to that block is cleared.
- To use the R_FlashProgramLockBit function, disable the FLASH_API_RX_CFG_IGNORE_LOCK_BITS definition in r_flash_api_rx_config.h.

3.9 R_FlashSetLockBitProtection

Enable or disable the lock bit protection functionality.

Format

```
uint8_t R_FlashSetLockBitProtection(uint32_t lock_bit);
```

Parameters

lock_bit Specifies whether lock bit protection functionality is enabled or disabled. A value of true or 1 or greater enables lock bit protection, and a value of false or 0 disables it.

Return Values

FLASH_SUCCESS: Normal end

FLASH_BUSY: Other processing of the flash memory is currently in progress.

Description

Enable or disable the lock bit protection functionality. When lock bit protection is enabled, it is not possible to write to or erase blocks for which the lock bit is set. When lock bit protection is disabled, it is possible to write to or erase all blocks, regardless of whether or not their lock bits are set.

Reentrant

No.

Example

```
uint8_t ret;

/* Enable lock bit protection (this is default out of reset) */
ret = R_FlashSetLockBitProtection(true);

/* Check for errors. */
if (FLASH_SUCCESS != ret)
{
    . . .
}
```

Special Notes

- If a block is erased while the lock bit protection functionality is disabled, the lock bit corresponding to that block is cleared.
- To use the R_FlashProgramLockBit function, disable the FLASH_API_RX_CFG_IGNORE_LOCK_BITS definition in r_flash_api_rx_config.h.

3.10 R_FlashGetStatus

Returns the API processing status.

Format

```
uint8_t R_FlashGetStatus(void);
```

Parameters

None.

Return Values

FLASH_SUCCESS: API functions can be run.

FLASH_BUSY: Other processing of the flash memory is currently in progress.

Description

This function can be used to check the API processing status in non-blocking mode.

Reentrant

Yes.

Example

```
uint8_t ret;

/* Blank check an entire data flash block. */
ret = R_FlashDataAreaBlankCheck(address, BLANK_CHECK_ENTIRE_BLOCK);

while( R_FlashGetStatus() == FLASH_BUSY )
{
    /* Wait for previous operation to finish. You could also stall this task
       and do some real work. */
}
```

Special Notes

None.

3.11 R_FlashCodeCopy

Copies the API code from the ROM section (PFRAM) to the RAM section (RPFRAM).

Format

```
    void R_FlashCodeCopy(void);
```

Parameters

None.

Return Values

None.

Description

Running the R_FlashCodeCopy function copies the API code from the ROM section (PFRAM) to the RAM section (RPFRAM).

Reentrant

Yes.

Example

```
/* Transfer Flash API code to RAM so that we can program/erase ROM. */
R_FlashCodeCopy();

/* Flash API can now program/erase ROM. */
```

Special Notes

- It is not necessary to run the R_FlashCodeCopy function if you instead copy the code by editing dbsct.c.
- To use the R_FlashCodeCopy function, enable the FLASH_API_RX_CFG_COPY_CODE_BY_API definition in r_flash_api_rx_config.h.

3.12 R_FlashGetVersion

Returns the version number of the API.

Format

```
    uint32_t R_FlashGetVersion(void);
```

Parameters

None.

Return Values

API version number

Description

The API version number is reported as a return value. The upper 2 bytes of the return value represent the major version number, and the lower 2 bytes represent the minor version number. For example, version 4.25 is represented as a return code of 0x00040019.

Reentrant

Yes.

Example

```
uint32_t cur_version;

/* Get version of installed Flash API. */
cur_version = R_FlashGetVersion();

/* Check to make sure version is new enough for this application's use. */
if (MIN_VERSION > cur_version)
{
    /* This Flash API version is not new enough and does not have XXX feature
       that is needed by this application. Alert user. */
    ....
}
```

Special Notes

R_FlashGetVersion function is defined as an inline function in the r_flash_api_rx.c file.

4. Reference Information

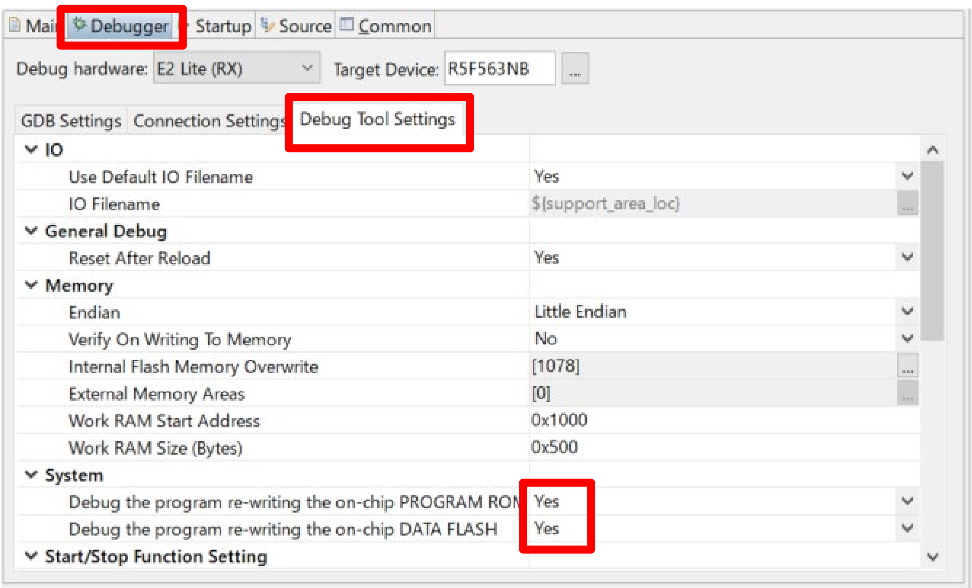
4.1 Emulator Debugging Settings

If you use the default debugging settings in the integrated development environment (e^2 studio or CS+), you will not be able to use the memory monitoring function to check the contents of the flash memory programmed by the API. It will be possible to program the flash memory, but the monitored values will not by updated by downloaded values.

If you want to be able to confirm the programmed data, change the debugger tool settings in the debugging configuration as shown below.

e^2 studio

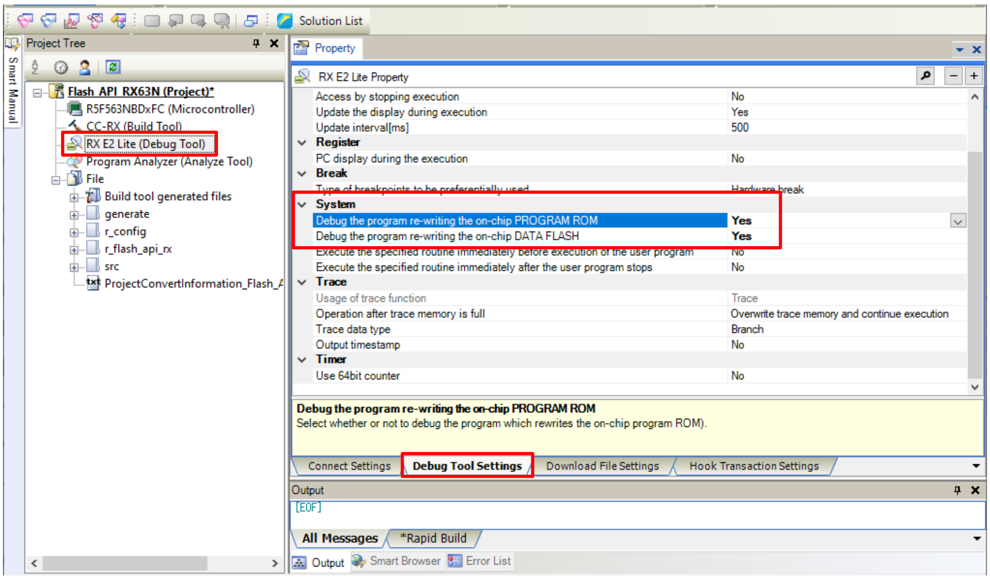
1. In Project Explorer, click the project to be debugged.
2. Click Run → Debug Configurations. . . to open the Debug Configurations window.
3. In the Debug Configurations window, open the **Renesas GDB Hardware Debugging** debugging configuration display, then click the debugging configuration of the debugging target.
4. Switch to the **Debugger** tab, click the **Debug Tool Settings** subtab of the **Debugger** tab, and enter the following settings.
   — System
   - Debug the program re-writing the on-chip PROGRAM ROM : Yes
   - Debug the program re-writing the on-chip DATA FLASH : Yes



— CS+

Select **RX E2 Lite(Debug Tool)** → **Debug Tool Settings** → **System**, and enter the following settings.

- Debug the program re-writing the on-chip PROGRAM ROM : Yes
- Debug the program re-writing the on-chip DATA FLASH : Yes



4.2 Using Flash Programmer to Read Programmed Data

In order to use Renesas Flash Programmer (RFP) to read the contents of flash memory programmed by a user program, you must set an ID code when you download the user program. The ID code is a security function built into RX Family devices. Note that if a connection is established without first setting an ID code, RFP will erase the entire flash memory area. For details of the ID code function, refer to the user’s manual of the specific device.

5. Sample Project

5.1 Overview

A sample project for each target device is provided as an accompaniment to this application note. The sample project is provided in e^2 studio project format, and it can be imported using e^2 studio or CS+ and used to confirm the operation of the API.

5.2 Operation Confirmation Environment

The operation of the sample project has been confirmed on the following environment.

Table 5.1 Operation Confirmation Environment

Item	Description
IDE	Renesas Electronics e^2 studio 2022-01
C compiler	Renesas Electronics C/C++ Compiler for RX Family V3.04.00 The compile options used are the default settings of the integrated development environment.
Endian order	Big endian/little endian
Operating mode	Single-chip mode
Processor mode	Supervisor mode
Board used	Renesas Starter Kit for RX610 (product No.: R0K55610xxxxxxx) Renesas Starter Kit for RX62G (product No.: R0K50562Gxxxxxx) Renesas Starter Kit+ for RX62N (product No.: R0K5562Nxxxxxxx) Renesas Starter Kit for RX62T (product No.: R0K5562Txxxxxxx) Renesas Starter Kit for RX630 (product No.: R0K505630xxxxxx) Renesas Starter Kit+ for RX63N (product No.: R0K50563Nxxxxxx) Renesas Starter Kit for RX63T (144-pin) (product No.: R0K5563THxxxxxx) Renesas Starter Kit for RX210 (B version) (product No.: R0K505210xxxxxx) Renesas Starter Kit for RX220 (product No.: R0K505220xxxxxx) Hokuto Denshi Co., Ltd. HSB Series Microcontroller Board (catalog number: HSBRX21AP-B)

5.3 Basic Operation of Sample Program

The program code of the sample project performs processing related to the settings of the configuration options. Table 5.2 summarizes the operation of the sample program. For details, refer to the source code of the sample program (flash_api_rx_demo_main.c).

Table 5.2 Operation of Sample Program

Function and Processing Overview	Function Configuration Options
flash_api_demo_df_tests function Uses API functions to perform processing, such as erasing, blank checking, and programming, on all the blocks in the data flash.	FLASH_API_RX_CFG_DATA_FLASH_BGO FLASH_API_RX_CFG_FLASH_TO_FLASH
flash_api_demo_rom_tests function Uses API functions to perform processing, such as erasing, programming, and lock bit setting, on all the blocks other than those in which the program is stored (16 KB)	FLASH_API_RX_CFG_ENABLE_ROM_PROGRAMMING FLASH_API_RX_CFG_ROM_BGO FLASH_API_RX_CFG_FLASH_TO_FLASH FLASH_API_RX_CFG_IGNORE_LOCK_BITS
flash_api_demo_rom_bgo_init function Performs processing to reassign the interrupt vector table to the RAM in non-blocking mode.	FLASH_API_RX_CFG_ROM_BGO
flash_api_demo_lock_bit_tests function Performs processing such as lock bit enable/disable and setting, and reading data.	FLASH_API_RX_CFG_IGNORE_LOCK_BITS

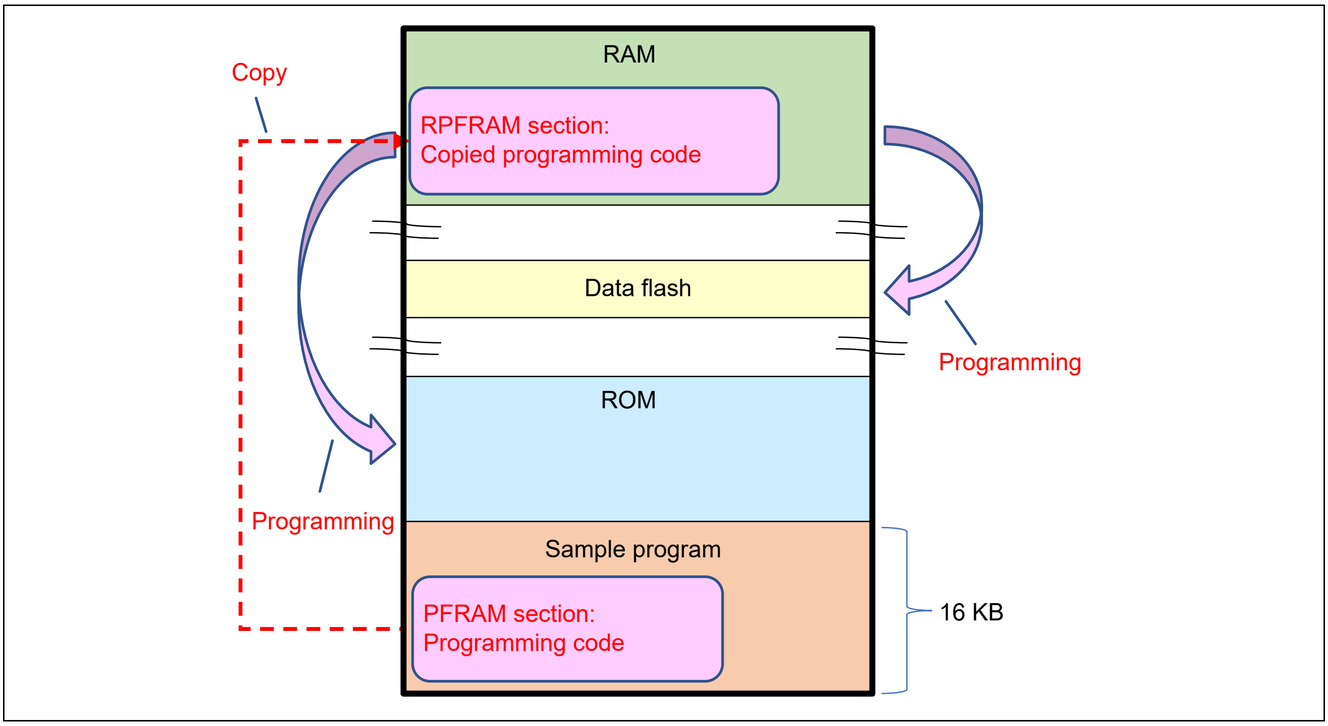


Figure 5.1 Sample Program Processing Overview (with ROM Programming Enabled)

5.4 Operating Clock Settings for Sample Project

The sample projects for some devices make use of code described in the associated “Initial Setting” application note to make operating clock settings. The operating clock settings used are the default settings listed in the “Initial Setting” application note. Refer to section 6, Reference Documents, for a listing of “Initial Setting” application notes.

The HardwareSetup function of generate/hwsetup.c performs operating clock settings in each sample project.

5.5 Importing a Project

The sample programs are distributed in e² studio project format. This section shows how to import a project into e² studio or CS+. After importing the sample project, make sure to confirm build and debugger setting.

5.5.1 Importing a Project into e² studio

To use sample programs in e² studio, follow the steps below to import them into e² studio. In projects managed by e2 studio, do not use space codes, multibyte characters, and symbols such as "$", "#", "%" in folder names or paths to them.

(Note that depending on the version of e² studio you are using, the interface may appear somewhat different from the screenshots below.)

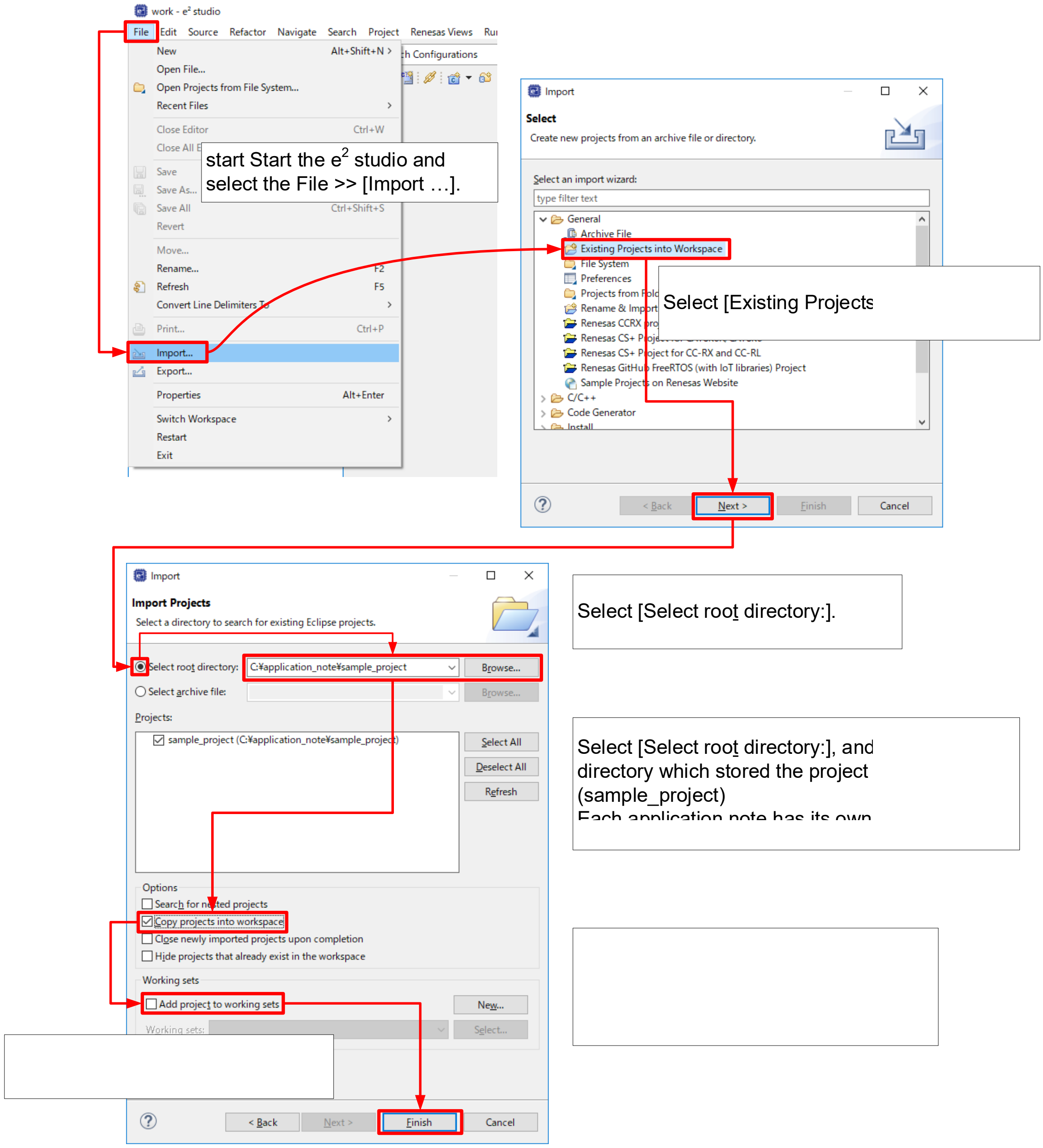


Figure 5.2 Importing a Project into e² studio

5.5.2 Importing a Project into CS+

To use sample programs in CS+, follow the steps below to import them into CS+. In projects managed by CS +, do not use space codes, multibyte characters, and symbols such as "$", "#", "%" in folder names or paths to them.

(Note that depending on the version of CS+ you are using, the interface may appear somewhat different from the screenshots below.)

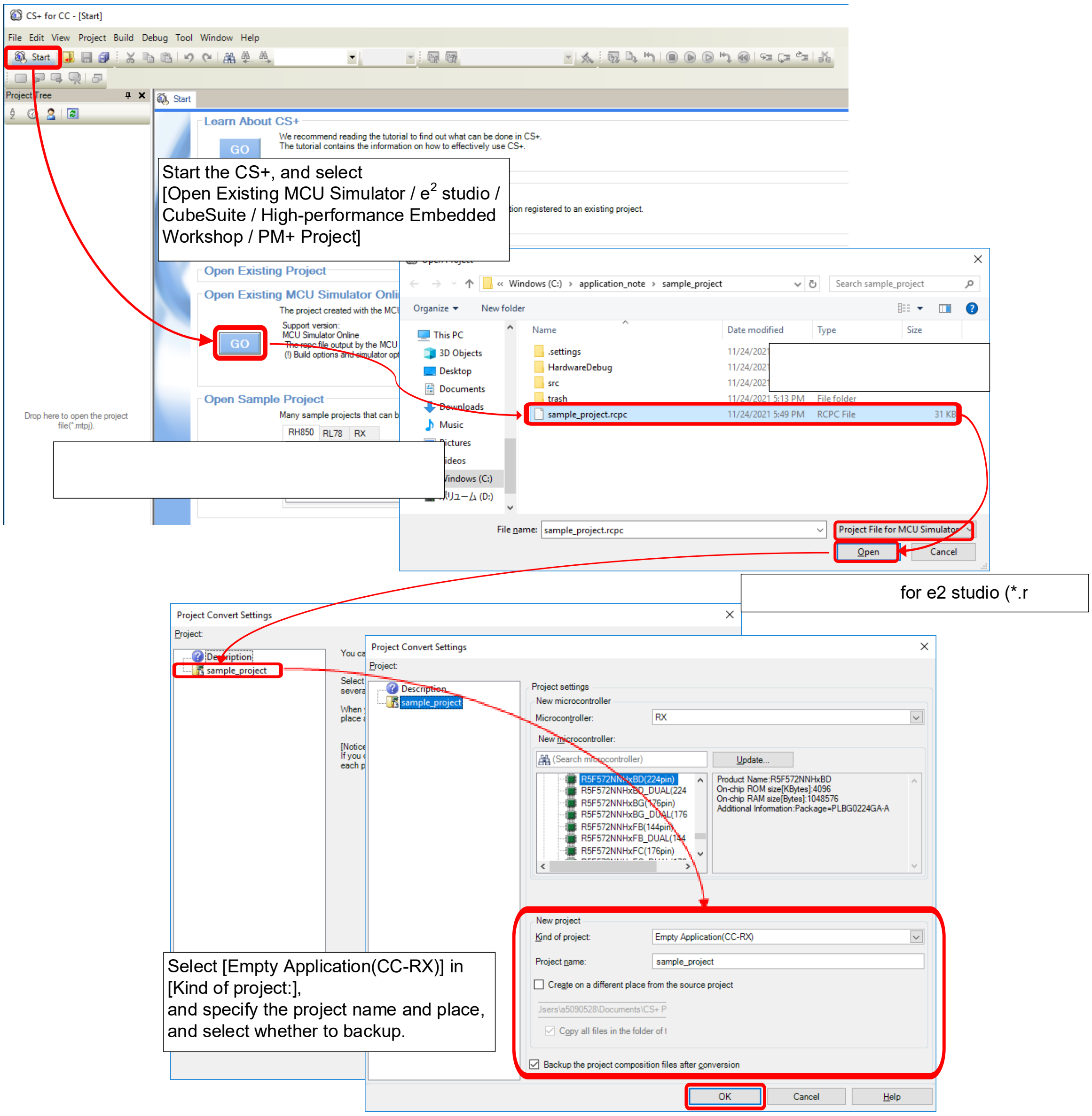


Figure 5.3 Importing a Project into CS+

6. Reference Documents

User's Manual: Hardware

Title	Revision	Document No.
RX610 Group User's Manual: Hardware	1.20	R01UH0032EJ0120
RX62N Group, RX621 Group User's Manual: Hardware	1.40	R01UH0033EJ0140
RX62T Group, RX62G Group User's Manual: Hardware	2.00	R01UH0034EJ0200
RX630 Group User's Manual: Hardware	1.60	R01UH0040EJ0160
RX63N Group, RX631 Group User's Manual: Hardware	1.80	R01UH0041EJ0180
RX63T Group User's Manual: Hardware	2.20	R01UH0238EJ0220
RX210 Group User's Manual: Hardware	1.50	R01UH0037EJ0150
RX220 Group User's Manual: Hardware	1.10	R01UH0292EJ0110
RX21A Group User's Manual: Hardware	1.10	R01UH0251EJ0110

Application Note

Title	Revision	Document No.
RX630 Group Initial Setting	1.00	R01AN1004EJ0100
RX63N Group, RX631 Group Initial Setting	1.10	R01AN1245EJ0110
RX63T Group Initialization Example	1.01	R01AN1252EJ0101
RX210 Group Initial Setting	2.21	R01AN1002EJ0221
RX220 Group Initial Setting	1.10	R01AN1494EJ0110
RX21A Group Initial Setting	1.10	R01AN1486EJ0110

Technical Update/Technical News
(The latest version can be downloaded from the Renesas Electronics website.)

User's Manual: Development environment
RX Family CC-RX Compiler User's Manual (R20UT3248)
(The latest version can be downloaded from the Renesas Electronics website.)

Revision History

Rev.	Date	Description	
		Page	Summary
1.00	Jan. 27, 2010	—	First edition issued
1.20	Feb. 11, 2010	—	Minor text revisions, and section on disabling interrupts added.
1.30	Mar. 5, 2010	—	Fixes based on recommendations from RTE.
1.40	May 26, 2010	—	Revised to include support for the RX62x Group.
1.41	Jun. 11, 2010	—	Fixed typographical errors, etc.
1.43	Feb. 18, 2011	—	Blank check function parameter description updated.
2.00	Apr. 27, 2011	—	Background operation (BGO), flash to flash transfer, and lock bit protection functionality added.
2.10	Jul. 11, 2011	—	Support for RX630, RX631, and RX63N devices added. DATA_FLASH_OPERATION_PIPL and ROM_OPERATION_PIPL definitions deleted, and section added explaining why this was done. Added R_FlashEraseRange() function to API. Section on ROM area boundaries (previously Section 3.4) rewritten to apply to RX610 and RX63x devices.
2.20	Mar. 27, 2012	—	Moved document over to new template. Restructured existing data and added new information about using r_bsp package. Added R_FlashCodeCopy() function to the API.
2.30	Sep. 12, 2012	—	Added R_FlashGetVersion() function to the API. Removed config macro for not using r_bsp because the code has been modified to recognize this automatically. Added "Configure for Only Data Flash Use," "Erase Entire User Application Area," "Reading from Data Flash After Reset," "Checking if a Data Flash Location is Blank (Erased)," and "Putting Flash API in User Boot Area" sections. Added blank check size table in R_FlashDataAreaBlankCheck section.
2.40	Jul. 1, 2013	—	Added support for RX210, RX62G, and RX63T devices. Since RX200 Series devices are now supported, changed name from "Simple Flash API for RX600" to "Simple Flash API for RX." Added "Checking if a Data Flash Location is Blank (Erased)" section, and added note on first page about where to find information about why erased data flash locations are not read as 0xFF. Added list of API functions to beginning of API Functions section. Added Demo Projects section.
2.50	Mar. 6, 2015	—	Added support for RX21A and RX220. For all instructions that referenced HEW, the equivalent steps are now provided for e^2 studio. Added "Execute from Data Flash" subsection. Added "Access Rules" subsection. Added "Related Documents" to cover page. Added "Memory Requirements" subsection. Removed "Bootloader Implementations" section. Revised "R_FlashDataAreaBlankCheck" subsection with use of new BLANK_CHECK_SMALLEST macro. Use of BLANK_CHECK_2_BYTE and BLANK_CHECK_8_BYTE is to be discontinued. Removed "Demo" section.

Rev.	Date	Description	
		Page	Summary
3.00	Feb.14, 2022	Overall	Changed configuration from one dependent on BSP to one not dependent on BSP. Added support for RX631 Group devices with ROM size of 256 KB.
		Document	Changed overall structure and contents of document. < Key points > • 2.4 Configuration The following new options have been added: — FLASH_MCU_xxxx — FLASH_API_RX_CFG_ICLK_HZ — FLASH_API_RX_CFG_FCLK_HZ — FLASH_API_RX_CFG_ROM_SIZE_BYTES — FLASH_API_RX_CFG_DATA_FLASH_SIZE_BYTES • 5. Sample Project Sections removed in Rev. 2.50 have been re-added.
		Program	Changed configuration from one dependent on r_bsp to one not dependent on r_bsp. • Added following new options to r_flash_api_rx_config.h. — FLASH_MCU_xxxx — FLASH_API_RX_CFG_ICLK_HZ — FLASH_API_RX_CFG_FCLK_HZ — FLASH_API_RX_CFG_ROM_SIZE_BYTES — FLASH_API_RX_CFG_DATA_FLASH_SIZE_BYTES • Definitions related to r_bsp (BSP_XXX) have been changed to new optional definitions in all source files and header files. • Specification changed to no longer use the r_bsp hardware locking mechanism. — The flash_grab_state function has been changed. — The flash_release_state function has been changed. • The r_flash_api_rx_if.h file has been deleted from the following definitions: — BLANK_CHECK_8_BYTE — BLANK_CHECK_2_BYTE Added support for RX631 Group devices with ROM size of 256 KB. • Added information on 256 KB products to r_flash_api_rx63n.h. Added countermeasures as described in Tool News (R20TS0805EJ0100).

General Precautions in the Handling of Microprocessing Unit and Microcontroller Unit Products

The following usage notes are applicable to all Microprocessing unit and Microcontroller unit products from Renesas. For detailed usage notes on the products covered by this document, refer to the relevant sections of the document as well as any technical updates that have been issued for the products.

1. Precaution against Electrostatic Discharge (ESD)
   A strong electrical field, when exposed to a CMOS device, can cause destruction of the gate oxide and ultimately degrade the device operation. Steps must be taken to stop the generation of static electricity as much as possible, and quickly dissipate it when it occurs. Environmental control must be adequate. When it is dry, a humidifier should be used. This is recommended to avoid using insulators that can easily build up static electricity. Semiconductor devices must be stored and transported in an anti-static container, static shielding bag or conductive material. All test and measurement tools including work benches and floors must be grounded. The operator must also be grounded using a wrist strap. Semiconductor devices must not be touched with bare hands. Similar precautions must be taken for printed circuit boards with mounted semiconductor devices.
2. Processing at power-on
   The state of the product is undefined at the time when power is supplied. The states of internal circuits in the LSI are indeterminate and the states of register settings and pins are undefined at the time when power is supplied. In a finished product where the reset signal is applied to the external reset pin, the states of pins are not guaranteed from the time when power is supplied until the reset process is completed. In a similar way, the states of pins in a product that is reset by an on-chip power-on reset function are not guaranteed from the time when power is supplied until the power reaches the level at which resetting is specified.
3. Input of signal during power-off state
   Do not input signals or an I/O pull-up power supply while the device is powered off. The current injection that results from input of such a signal or I/O pull-up power supply may cause malfunction and the abnormal current that passes in the device at this time may cause degradation of internal elements. Follow the guideline for input signal during power-off state as described in your product documentation.
4. Handling of unused pins
   Handle unused pins in accordance with the directions given under handling of unused pins in the manual. The input pins of CMOS products are generally in the high-impedance state. In operation with an unused pin in the open-circuit state, extra electromagnetic noise is induced in the vicinity of the LSI, an associated shoot-through current flows internally, and malfunctions occur due to the false recognition of the pin state as an input signal become possible.
5. Clock signals
   After applying a reset, only release the reset line after the operating clock signal becomes stable. When switching the clock signal during program execution, wait until the target clock signal is stabilized. When the clock signal is generated with an external resonator or from an external oscillator during a reset, ensure that the reset line is only released after full stabilization of the clock signal. Additionally, when switching to a clock signal produced with an external resonator or by an external oscillator while program execution is in progress, wait until the target clock signal is stable.
6. Voltage application waveform at input pin
   Waveform distortion due to input noise or a reflected wave may cause malfunction. If the input of the CMOS device stays in the area between V_{IL} (Max.) and V_{IH} (Min.) due to noise, for example, the device may malfunction. Take care to prevent chattering noise from entering the device when the input level is fixed, and also in the transition period when the input level passes through the area between V_{IL} (Max.) and V_{IH} (Min.).
7. Prohibition of access to reserved addresses
   Access to reserved addresses is prohibited. The reserved addresses are provided for possible future expansion of functions. Do not access these addresses as the correct operation of the LSI is not guaranteed.
8. Differences between products
   Before changing from one product to another, for example to a product with a different part number, confirm that the change will not lead to problems. The characteristics of a microprocessing unit or microcontroller unit products in the same group but having a different part number might differ in terms of internal memory capacity, layout pattern, and other factors, which can affect the ranges of electrical characteristics, such as characteristic values, operating margins, immunity to noise, and amount of radiated noise. When changing to a product with a different part number, implement a system-evaluation test for the given product.

Notice

1. Descriptions of circuits, software and other related information in this document are provided only to illustrate the operation of semiconductor products and application examples. You are fully responsible for the incorporation or any other use of the circuits, software, and information in the design of your product or system. Renesas Electronics disclaims any and all liability for any losses and damages incurred by you or third parties arising from the use of these circuits, software, or information.
2. Renesas Electronics hereby expressly disclaims any warranties against and liability for infringement or any other claims involving patents, copyrights, or other intellectual property rights of third parties, by or arising from the use of Renesas Electronics products or technical information described in this document, including but not limited to, the product data, drawings, charts, programs, algorithms, and application examples.
3. No license, express, implied or otherwise, is granted hereby under any patents, copyrights or other intellectual property rights of Renesas Electronics or others.
4. You shall be responsible for determining what licenses are required from any third parties, and obtaining such licenses for the lawful import, export, manufacture, sales, utilization, distribution or other disposal of any products incorporating Renesas Electronics products, if required.
5. You shall not alter, modify, copy, or reverse engineer any Renesas Electronics product, whether in whole or in part. Renesas Electronics disclaims any and all liability for any losses or damages incurred by you or third parties arising from such alteration, modification, copying or reverse engineering.
6. Renesas Electronics products are classified according to the following two quality grades: “Standard” and “High Quality”. The intended applications for each Renesas Electronics product depends on the product’s quality grade, as indicated below.
   "Standard": Computers; office equipment; communications equipment; test and measurement equipment; audio and visual equipment; home electronic appliances; machine tools; personal electronic equipment; industrial robots; etc.
   "High Quality": Transportation equipment (automobiles, trains, ships, etc.); traffic control (traffic lights); large-scale communication equipment; key financial terminal systems; safety control equipment; etc.
   Unless expressly designated as a high reliability product or a product for harsh environments in a Renesas Electronics data sheet or other Renesas Electronics document, Renesas Electronics products are not intended or authorized for use in products or systems that may pose a direct threat to human life or bodily injury (artificial life support devices or systems; surgical implantations; etc.), or may cause serious property damage (space system; undersea repeaters; nuclear power control systems; aircraft control systems; key plant systems; military equipment; etc.). Renesas Electronics disclaims any and all liability for any damages or losses incurred by you or any third parties arising from the use of any Renesas Electronics product that is inconsistent with any Renesas Electronics data sheet, user’s manual or other Renesas Electronics document.
7. No semiconductor product is absolutely secure. Notwithstanding any security measures or features that may be implemented in Renesas Electronics hardware or software products, Renesas Electronics shall have absolutely no liability arising out of any vulnerability or security breach, including but not limited to any unauthorized access to or use of a Renesas Electronics product or a system that uses a Renesas Electronics product. RENESAS ELECTRONICS DOES NOT WARRANT OR GUARANTEE THAT RENESAS ELECTRONICS PRODUCTS, OR ANY SYSTEMS CREATED USING RENESAS ELECTRONICS PRODUCTS WILL BE INVULNERABLE OR FREE FROM CORRUPTION, ATTACK, VIRUSES, INTERFERENCE, HACKING, DATA LOSS OR THEFT, OR OTHER SECURITY INTRUSION (“Vulnerability Issues”). RENESAS ELECTRONICS DISCLAIMS ANY AND ALL RESPONSIBILITY OR LIABILITY ARISING FROM OR RELATED TO ANY VULNERABILITY ISSUES. FURTHERMORE, TO THE EXTENT PERMITTED BY APPLICABLE LAW, RENESAS ELECTRONICS DISCLAIMS ANY AND ALL WARRANTIES, EXPRESS OR IMPLIED, WITH RESPECT TO THIS DOCUMENT AND ANY RELATED OR ACCOMPANYING SOFTWARE OR HARDWARE, INCLUDING BUT NOT LIMITED TO THE IMPLIED WARRANTIES OF MERCHANTABILITY, OR FITNESS FOR A PARTICULAR PURPOSE.
8. When using Renesas Electronics products, refer to the latest product information (data sheets, user’s manuals, application notes, “General Notes for Handling and Using Semiconductor Devices” in the reliability handbook, etc.), and ensure that usage conditions are within the ranges specified by Renesas Electronics with respect to maximum ratings, operating power supply voltage range, heat dissipation characteristics, installation, etc. Renesas Electronics disclaims any and all liability for any malfunctions, failure or accident arising out of the use of Renesas Electronics products outside of such specified ranges.
9. Although Renesas Electronics endeavors to improve the quality and reliability of Renesas Electronics products, semiconductor products have specific characteristics, such as the occurrence of failure at a certain rate and malfunctions under certain use conditions. Unless designated as a high reliability product or a product for harsh environments in a Renesas Electronics data sheet or other Renesas Electronics document, Renesas Electronics products are not subject to radiation resistance design. You are responsible for implementing safety measures to guard against the possibility of bodily injury, injury or damage caused by fire, and/or danger to the public in the event of a failure or malfunction of Renesas Electronics products, such as safety design for hardware and software, including but not limited to redundancy, fire control and malfunction prevention, appropriate treatment for aging degradation or any other appropriate measures. Because the evaluation of microcomputer software alone is very difficult and impractical, you are responsible for evaluating the safety of the final products or systems manufactured by you.
10. Please contact a Renesas Electronics sales office for details as to environmental matters such as the environmental compatibility of each Renesas Electronics product. You are responsible for carefully and sufficiently investigating applicable laws and regulations that regulate the inclusion or use of controlled substances, including without limitation, the EU RoHS Directive, and using Renesas Electronics products in compliance with all these applicable laws and regulations. Renesas Electronics disclaims any and all liability for damages or losses occurring as a result of your noncompliance with applicable laws and regulations.
11. Renesas Electronics products and technologies shall not be used for or incorporated into any products or systems whose manufacture, use, or sale is prohibited under any applicable domestic or foreign laws or regulations. You shall comply with any applicable export control laws and regulations promulgated and administered by the governments of any countries asserting jurisdiction over the parties or transactions.
12. It is the responsibility of the buyer or distributor of Renesas Electronics products, or any other party who distributes, disposes of, or otherwise sells or transfers the product to a third party, to notify such third party in advance of the contents and conditions set forth in this document.
13. This document shall not be reprinted, reproduced or duplicated in any form, in whole or in part, without prior written consent of Renesas Electronics.
14. Please contact a Renesas Electronics sales office if you have any questions regarding the information contained in this document or Renesas Electronics products.

(Note1) “Renesas Electronics” as used in this document means Renesas Electronics Corporation and also includes its directly or indirectly controlled subsidiaries.

(Note2) “Renesas Electronics product(s)” means any product developed or manufactured by or for Renesas Electronics.

(Rev.5.0-1 October 2020)

Corporate Headquarters

TOYOSU FORESIA, 3-2-24 Toyosu,
Koto-ku, Tokyo 135-0061, Japan
www.renesas.com

Trademarks

Renesas and the Renesas logo are trademarks of Renesas Electronics Corporation. All trademarks and registered trademarks are the property of their respective owners.

Contact information

For further information on a product, technology, the most up-to-date version of a document, or your nearest sales office, please visit: www.renesas.com/contact/.